Лабораторная работа №2. Классы, интерфейсы, исключения

Теоретическая часть

Классы и интерфейсы

Класс – тип данных, определяемый пользователем и предназначенный для создания объектов.

Объекты создаётся из класса с помощью оператора new. Каждый объект обладает опредёлённым состоянием и поведением. Используя интерфейс объекта, можно управлять им и определять его состояние. Через интерфейсы объекты могут взаимодействовать друг с другом. Использование объектов осуществляется посредством ссылок на эти объекты или с помощью интерфейсных ссылок. Интерфейсная ссылка ограничивают применение открытых функций объекта принадлежащих только интерфейсу этой ссылке. Класс может наследовать множество интерфейсов, для каждого из которых можно создать свою интерфейсную ссылку, разделив тем самым при необходимости доступ к функциям объекта.

Класс и наследование

Класс определяется следующим образом:

[**public**] **class** имя-класса
[{**extends** базовый класс [, **implements** имя-интерф] . . .]}
|{ **implements** имя-интерф [,имя-интерф] . . .}]
{
  объявление данных и описание функций, свойств и событий
}

В простейшем виде определение класса начинается с ключевого слова **class**, затем следует имя класса и тело, содержащее объявление данных и функций. По умолчанию все эти члены класса имеют доступ «в рамках пакета» и могут использоваться применительно к объекту этого класса через ссылку к этому объекту.

Некоторые члены класса, которые применяются только внутри объекты и которые нельзя использовать вне объекта, помечаются доступом private (protected для доступа только в рамках пакета и из предков).

Ключевое слово public перед словом class делает класс доступным пользователям-программистам.

Класс (sub class) может наследовать (extends) один базовый класс (superclass), что позволяет в данном классе воспользоваться членами наследуемого базового класса, имеющими доступ public или protected. Специальное ключевое слово super используется для вызова конструктора базового класса и может использоваться для разрешения неоднозначности при ссылке к доступным членам базового класса.

В нижеследующем примере, поясняющем наследование класса SubClass из класса SuperClass, надлежит применить слово super при вызове конструктора базового класса, а также в теле функции getB() класса SubClass, вызывающей функцию getB() базового класса, чтобы вызвать функцию не основного, а базового класса.

```
class Dan {
  public int xSup, xSub;
  public boolean b;
  public Dan (int Xsub, int Xsup, boolean B) {xSub= Xsub; xSup= Xsup; b= B;}
```

```
}

class SuperClass {
  int x;
  private boolean b;
  public SuperClass (int X, boolean B) {x= X; b= B;}
  public boolean getB () {return b;}
}

class SubClass extends SuperClass {// Подкласс (sub class)
  int x;
  public SubClass (int Xsup, int Xsub, boolean B) // Конструктор
  {
    super (Xsub, B); // Вызов конструктора базового класса
    x= Xsup;
  }
  // Получить объект типа Dan с данными
  public Dan get () {return new Dan(x, super.x, super.getB());}
}

class Test {
  public static void main (String[] args) {
    SuperClass supC= new SuperClass (5, true );// Создать объект суперкласса
    System.out.println ("supC.x= "+ supC.x + " supC.b= " + supC.getB());
    // Создать объект подкласса
    SubClass subC= new SubClass (55, 555, false );
    // Получить объект данных
    Dan d= subC.get();
    System.out.println ("subC.x= "+ d.xSub + " supC.x= "+ d.xSup + " subC.b= " + d.b);
  }
}

//Result:
supC.x= 5 supC.b= true
subC.x= 55 supC.x= 555 subC.b= false
```

В примере показана передача одновременно всех данных объекта класса SupClass. Поскольку в языке Java в функциях осуществляется передача параметров примитивных типов только по значению, в класс SupClass включена функция get(), возвращающая ссылку на объект типа Dan, содержащий значения данных объекта класса SupClass.

О функциях класса

Функции описываются только в классах и часто называются методами.

Различают обычные функции класса и статические функции. Описанию статической функции предшествует слово static. Обычные функции класса используются применительно к объекту класса через ссылку, а статические – применительно к классу через имя класса (один из случаев использования статических переменных).

Удобны перегруженные (overloading) функции, которые имеют одно и тоже имя, но разное количество или различные типы параметров. Перегруженные конструкторы позволяют разнообразно инициализировать объект при его создании.

В функциях языка Java аргументы передаются по значению или по ссылке. Типы boolean, char, byte, short, int, long, float, double передаются по значению, при этом копируется значение аргумента в параметр.

Если аргументом является объект, то он передаётся по ссылке. Здесь в параметр копируется ссылка, которая позволяет как передать данные объекта в функцию, так и модифицировать их из функции. Для иллюстрации передачи данных через параметр-ссылку включим в класс SubClass дополнительно перегруженную функцию get (Dan d):

```
public void get (Dan d) {d.xSub= x; d.xSup= super.x; d.b= super.getB();}
```

А в функции main() применим эту функцию следующим образом:

```
Dan d1= new Dan (0, 0, true);
subC.get(d1);
System.out.println ("subC.x= "+ d1.xSub +" supC.x= "+ d1.xSup + " subC.b= " + d1.b);
```

Как видим, применение прежней перегруженной функции get() более удобно. Открытые функции и данные суперкласса доступны в подклассе. Если желательно переопределить в подклассе некоторую функцию FuncSup() его подкласса, например, дополнить её выводом на консоль имени SubClass подкласса, то в теле переопределяемой функции подкласса надлежит вызвать одноимённую функцию суперкласса, воспользовавшись ключевым словом super:

```
void FuncSup () {super. FuncSup (); System.out.println (“SubClass”);}
```

Конструктор подкласса должен инициализировать данные суперкласса, вызвав конструктор суперкласса (первой строчкой), воспользовавшись ключевым словом super, и передав ему требуемые значения аргументов.

Интерфейсы

Интерфейс – это совокупность объявлений функций и статических переменных, доступных для применения и управления объектом извне. В отличие от класса из интерфейса нельзя создать объект. Интерфейсы для этого не предназначены. Их назначение – обязать класс, который наследует интерфейсы, обязательно реализовать функции этих интерфейсы для применения в объектах этого класса. Таким образом, интерфейс навязывает объекту некоторые правила поведения. Наследование классами некоторого интерфейса обязывает их объекты в схожих ситуациях вести себя одинаково, то есть использовать одни и те же функции. Например, наблюдаемые объекты и наблюдатели используют определённый интерфейс. Также соответствующие интерфейсы используют управляющие интерфейсные элементы, при этом стандартизируется реализация реакций на действия с этими элементами.

Интерфейс имеет следующую грамматику:

```
[public interface имя-интерф [extends имя-базов-интерф
   [   ,   имя-базов-интерф ] ... ]
 {
  объявление функций и переменных
 }
```

Ключевое слово public перед словом interface делает интерфейс доступным вне пакета (пользователю-программисту), в котором он определяется.

Следующий пример иллюстрирует создание и применение интерфейсов.

```
interface IFunc {
 public int getF();
}

interface IConst {
 public static final int verConst= 100;
}

class ClassInt implements IFunc, IConst {
 public int getF (){ return verConst;}
}

class TestInterface {
 public static void main (String[] args) {
  ClassInt cI= new ClassInt();
  System.out.println ("verConst= " + cI.getF());
  IFunc iF= cI;
  IConst iC= cI;
  System.out.println ("verConst= " + iF.getF());
  // System.out.println ("verConst= " + iC.getF()); // Error
  // Cannot find 'getF()' in 'IConst'
  System.out.println ("verConst= " + iC.verConst);
  System.out.println ("verConst= " + ClassInt.verConst);
 }
}

/* Result:
verConst= 100
verConst= 100
verConst= 100
verConst= 100
*/
```

В примере интерфейсы включают открытые (public) по умолчанию объявление функции и статическую final переменную. Наследующий интерфейс класс не обязан объявлять переменные интерфейса – в этом случае они будут переопределены и нести иной смысл. Переменная интерфейса статическая и поэтому может непосредственно применяться в классе. Более того, она явлена final и её значение не может быть изменено.

Интерфейс может наследовать другие интерфейсы, при этом належит применить ключевое слово extends. Наследование интерфейсов одноимёнными функциями и переменными может вызвать неоднозначность при их применении. Неоднозначность разрешается применением имён интерфейсов. Проиллюстрируем применение интерфейсов.

```
/* Интерфейсы и неоднозначность*/
interface IConst1 {
 static final int verConst= 101;
}

interface IConst2 {
 int verConst= 102;
```

```
}

interface IConst extends IConst1, IConst2 {
 static final int verConst= 100;
 int get (boolean b);
}

class ClassInt implements IConst {
 public int get (boolean b){ return b ? IConst1.verConst : IConst2.verConst;}
}

class TestInterface2 {
 public static void main (String[] args) {
  ClassInt cI= new ClassInt();
  System.out.print ("cI.verConst= " + cI.verConst);
  System.out.println (" ClassInt.verConst= " + ClassInt.verConst);
  //-----
  IConst1 iC1= cI;
  System.out.print ("iC1.verConst= " + iC1.verConst);
  IConst2 iC2= cI;
  System.out.print (" iC2.verConst= " + iC2.verConst);
  IConst iC= cI;
  System.out.println (" iC.verConst= " + iC.verConst);
  //--------------
  System.out.print ("IConst1.verConst= " + IConst1.verConst);
  System.out.print (" IConst2.verConst= " + IConst2.verConst);
  System.out.println (" IConst.verConst= " + IConst.verConst);
  //--------------
  System.out.println ("cI.get(true)= " + cI.get(true) + " cI.get(false)= "+ cI.get(false));
 }
}
/*Result:
cI.verConst= 100 ClassInt.verConst= 100
iC1.verConst= 101 iC2.verConst= 102 iC.verConst= 100
IConst1.verConst= 101 IConst2.verConst= 102 IConst.verConst= 100
cI.get(true)= 101 cI.get(false)= 102
*/
```

Обработка исключений

Обработка исключений, возникающих при выполнении программы, обеспечивает надёжность функционирования программы со стороны пользователя, поскольку предвиденные опытным программистом аварийные ситуации предупреждаются и обрабатываются. Программу спасает в критические моменты throw-and-catch концепция языка Java. При обработке исключений применяются ключевые слова try, catch, throw, throws и finally.

Блок обработки исключений имеет следующую грамматику:

```
try{контролируемый код}
{ catch (ExceptionType e) { обработка исключения } } . . .
[ finally { обработка выхода из блока try } ]
```

try-блок включает программный код, который должен контролироваться, поскольку может выбросить (throws) исключение (объект типа ExceptionType). Ниже расположенные catch-блоки перехватывают исключение (объект типа ExceptionType) и обрабатывают его. Исключение обрабатывается только в том случае, если присутствует соответствующий этому исключению catch-блок. Иначе программа завершается аварийно.

finally-блок, если присутствует, обрабатывается в любом случае после выполнения try-блок. Он незаменим, если при завершении программа должна освободить ресурсы.

Язык Java для ряда операций требует обязательной обработки исключений, например, при операциях ввода-вывода и использовании функции sleep() класса Thread. Отсутствие таких исключений выявляется компилятором.

Контролируемые коды могут включать дополнительные обработки исключений. В этом случае при возникновении исключений они обрабатываются по правилам стека.

Нижеприведённый пример иллюстрирует перехват исключения при делении на нуль.

```
class TestExc {
 static void Func ( ) { int m=10, n=0; n= m/n; }
 public static void main (String[] args) {
  try {
    Func ();}
  catch (ArithmeticException e) {
    System.out.println ("ArithmeticException happened");}
  catch (Exception e) {
    System.out.println ("Exception happened");}
  finally {
    System.out.println ("finally");}
 }
}
/* Result:
ArithmeticException happened
finally
*/
```

В примере применены два catch-блока. Первый catch-блок включает подкласс ArithmeticException суперкласса Exception. Компилятор обязывает размещать классы подклассы исключений перед их суперклассами.

Последовательность catch-блока должна предвидеть все возможные исключения в контролируемом программном блоке. При наличии исключения контролируемый блок выполняется частично, его выполнение прерывается и затем выполняется соответствующий catch-блок и finally-блок.

В последовательности catch-блоков кроме суперкласса Exception можно применить его подклассы ArithmeticException, ArrayIndexOutOfBoundsException, ArrayStoreException, ClassCastException, IllegalArgumentException, IllegalMonitorStateException, IllegalStateException, IllegalThreadStateException, IndexOutOfBoundsException, NegativeArraySizeException, NullPointerException, NumberFormatException, SecurityException, StringIndexOutOfBoundsException, UnsupportedOperationException.

Оператор throw и ключевое слово throws позволяет при необходимости генерировать своё исключение и обработать его по общим правилам. Например, если пользователь вводит в текстовые редакторы строки и функции преобразования их в числа выявляют, что вводятся не числа, то выбрасывается исключение.

Следующий пример иллюстрирует выброс и перехват собственного исключения.

```
class MyException extends Exception {
```

```
 //public MyException () {}
 public String toString () { return "MyException happened";}
}

class TestMyExc {
 static void Func() throws MyException {
  int m = 0, n = 10;
  if((m/n) == 0) throw new MyException ();
  System.out.print ("Next");
 }
 public static void main(String[] args) {
  try
   {Func ();}
  catch (MyException e) {System.out.println (e);}
 }
}
/*Result:
MyException happened
*/
```

В примере включён собственный класс MyException исключения, наследующий суперкласс Exception. В классе обязательно должна быть переопределена функция toString(), которая при возникновении исключения сообщит о нём. Конструктор может включать параметры, которым присваиваются аргументы, фиксирующие места выброса исключения. Эти параметры используются в телефункции toString() для выдачи более подробной информации об исключении.

Функция, в которой выбрасывается (throw) объект собственного исключения в заголовке должна содержать ключевое слово throws с перечнем используемых собственных исключений. В примере функция Func(), выбрасывающая исключение MyException, содержит в заголовке throws MyException. Эта функция вызывается в try-блоке, а нижеследующий catch-блок содержит обработку собственного исключения MyException.

Практическая часть

1. Ознакомится с теоретической частью.
2. Запустить примеры программ из теоретической части (продемонстрировать преподавателю).
3. Получить задание. Изучив полученное задание, записать наименования и назначение функций, которые должны быть реализованы в ходе разработки требуемого класса (согласовать с преподавателем).
4. На основе п.3 описать два интерфейса, которые должны содержать необходимые для последующей реализации класса функции и константы. Если в задании фигурируют какие-либо константы, то они должны находиться в описании интерфейсов.
5. Описать класс, осуществляющий интерфейсы из п.3.
6. При осуществлении функций разрабатываемого класса нужно описать собственные классы обработчики исключительных ситуаций, а также функции, которые будут выбрасывать собственные исключения. Варианты исключительных ситуаций приведены ниже в таблице №1. По согласованию с преподавателем допускается разработка своих предложенных на обсуждение исключительных ситуаций.

7. Показ использования функций класса осуществить созданием объектов класса в функции public static void main(String[] args) главного класса и вызовом функций этих объектов. При показе следует в ряде примеров предъявлять факт выбрасывания собственных исключений.
8. Включить в отчёт по лабораторной работе исходные данные тестов использования функций класса и соответствующие им результаты вывода на консоль.

Варианты заданий

Выполнить задание из первой лабораторной работы, но чтобы при этом в описании требуемого класса учитывались указанные исключительные ситуации (см. таблицу №2).

Таблица №1

№№	**Исключительная ситуация**
1.	В массиве число элементов меньше указанного
2.	В строке отсутствует какой-то символ
3.	Больше, чем некоторое число
4.	В массиве число элементов больше указанного
5.	Меньше, чем некоторое число
6.	Строка содержит некоторый символ
7.	В массиве число элементов равно указанному
8.	Равенство некоторому числу
9.	В строке есть литеры

Номер отличного от нуля разряда в таблице №2 соответствует номеру исключительной ситуации в таблице №1, которая должна быть осуществлена в разрабатываемом классе. Так, например, вариант №1 из таблицы №2 соответствует следующим исключительным ситуациям: «В массиве число элементов меньше указанного», «В строке отсутствует какой-то символ» и «Больше, чем некоторое число».

Таблица №2

№	**Номера исключительных ситуаций**
1.	000000111
2.	000001011
3.	000001101
4.	000001110
5.	000010011
6.	000010101
7.	000010110
8.	000011001
9.	000011010
10.	000011100
11.	000100011
12.	000100101
13.	000100110
14.	000101001
15.	000101010
16.	000101100
17.	000110001
18.	000110010
19.	000110100
20.	000111000
21.	001000011
22.	001000101
23.	001000110
24.	001001001
25.	001001010
26.	001001100
27.	001010001

28.	001010010
29.	001010100
30.	001011000
31.	001100001
32.	001100010
33.	001100100
34.	001101000
35.	001110000
36.	010000011
37.	010000101
38.	010000110
39.	010001001
40.	010001010
41.	010001100
42.	010010001
43.	010010010
44.	010010100
45.	010011000
46.	010100001
47.	010100010
48.	010100100
49.	010101000
50.	010110000
51.	011000001
52.	011000010
53.	011000100
54.	011001000
55.	011010000
56.	011100000
57.	100000011
58.	100000101
59.	100000110
60.	100001001
61.	100001010
62.	100001100
63.	100010001
64.	100010010
65.	100010100
66.	100011000
67.	100100001
68.	100100010
69.	100100100
70.	100101000
71.	100110000
72.	101000001
73.	101000010
74.	101000100
75.	101001000
76.	101010000
77.	101100000

78.	110000001
79.	110000010
80.	110000100
81.	110001000
82.	110010000
83.	110100000
84.	111000000